

第4章

いかにしてバグの原因を突き止めるか

――新人技術者のためのロジカル・シンキング入門(1)

冴木 元

ここでは，大規模な組み込みシステム開発におけるデバッグの考えかたを解説する．自分が開発に携わっている組み込みシステムでバグが発生したときに重要なのは，「問題の切り分け」である．この切り分けをスムーズに行う方法はある程度パターン化できる．そのパターンを押さえていれば，いろいろなシステムのデバッグに適用できる．（編集部）

Aさんは，とあるメーカの研究所で生体認証にかかわる認識アルゴリズムの研究開発に携わっています．

● ある日突然システムがハングアップ！ さてどうする？

Aさんらは虹彩（瞳孔の周りの輪状の膜）を使った生体認証アルゴリズムを開発しました．この方法は，従来のものと比べて認識の誤差が格段に小さいという特徴を持っているため，商品化には社内でも大きな期待がかけられていました．

このアルゴリズムは，人の出入りをチェックする認証装置に採用されることになっています．施設を利用する人はあらかじめ虹彩のパターンを登録しておきます．施設に入るときには虹彩を小さなカメラに映し，登録済みのパターンと一致していれば入場できる，というものです．

開発も終盤にさしかかったとき，異変が起きました．テスト中の認証装置がときどきハングアップしてしまい，電源をいったん落とさないと元に戻らないというのです．認識モジュールの実装にもかかわってきたAさんにも，当然火の粉が降りかかってきました．Aさんはどのようにして問題の切り分けを進めるべきでしょうか？

1 現象を再現させる

パソコンのような汎用的なコンピュータを用いずに，独自のハードウェア（例えば携帯電話や家電製品）に搭載するためのソフトウェアを「組み込みソフトウェア」と呼びます．最近では組み込みソフトウェアを搭載した製品としていろいろなものが開発されていて，ヒット商品も多々あります．身の回りのありとあらゆるところに組み込みソフトウェアを用いた製品が存在する現象を指して，「ユビキタス（遍在する，という意味）」などと呼ぶこともあります．

最近の組み込みソフトウェアの特徴は，昔と比べて格段に機能が複雑になり，規模が増大していることでしょう．携帯電話などはあの小ささで数十Mステップにも及ぶソフトウェアを使っている製品もあるくらいです．この規模のソフトウェアを安定して動かそうとすると，20年ほど前ならまちがいなく大型コンピュータが必要だったはずです．

では，このような巨大な組み込みシステムで問題が発生

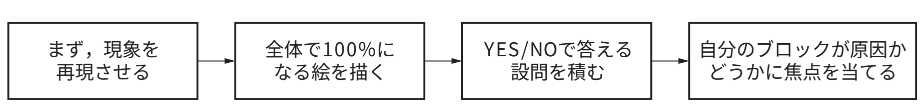


図1
システムをデバッグするには
結合された組み込みシステムをデバッグするための枠組みを示す．巨大システムのデバッグには，無視してはならない一定の手順がある．バグの現象が深刻なときこそ，秩序だった切り分けが必要不可欠となる．個々の内容は後で詳しく述べる．

KeyWord
生体認証，アルゴリズム，組み込みソフトウェア，デバッグ，メモリ・リーク，malloc，リアルタイム処理，DMA，ダミー・ルーチン，リファレンス・モデル

した場合，どのようにして問題の切り分けを行えばよいのでしょうか？ ここでは，ハードウェアやシステムに関する具体的な知識はさておき，問題の切り分けの考えかたそのものを検討してみたいと思います(**図1**).

● まずはシステム全体での「現象の再現」がポイント

バグ報告が上がってきたとき，開発者はまず何を真っ先に行うべきでしょうか？ 筆者なら，何をおいてもまず現象の再現を試みます．この段階では単体レベルでの再現，すなわち認証モジュールがモジュールとしてハングアップするかどうかをいきなり試すのではなく，システム全体としての現象を再現させます.

システム全体としての現象の再現は，かならずしも製品を用いて行われるとはかぎりません．ハードウェア・レベルでデバッグしやすいように，製品とは別に評価用ボードを用意する場合も多いでしょうから，そこでの再現も含まれます.

このときたいせつなのは，再現条件をよく吟味せずに「問題が起こるはずはない」などといって，自分の担当部分に問題がないことを証明しようとしないことです(**図2**)．チームによる開発では技術者どうしの連携が重要です．一つの機能ブロックの担当者が訳のわからないことを言って問題の切り分けに協力しなければ，調和が乱れてしまいます．これは単なるコミュニケーションだけの問題ではありません．この後で詳しく述べますが，現象の再現を怠ってみずからの無罪を証明しようとするアプローチでは，根本原因の特定が遅れ，技術的な観点から見ても問題が出てくるのです.

巨大な組み込みシステムでは，全体から部分へスムーズに移行することが問題の切り分けのポイントになります．上述したように，すべての切り分けに先立つものとして現象の再現が重要となります.

組み込みシステムの中には，重さ数t(トン)もある交換機のように簡単に持ち運びできないものもあるため，この議論がどのような場合でも簡単に実行に移せるわけではありません．しかし，評価ボードを使う場合も含めて，開発の責任者はできるかぎり製品の実環境を入手することに努めるべきだと筆者は考えます．顧客から貸与してもらうとか，あるいは一定期間ほかの開発チームといっしょにテストを行っている工場に集まるとか，いろいろな方法があります．とくにシステムが巨大な場合，エラーの再現を自分の目で確かめられるかどうかがデバッグの効率を大きく左右します.

なお，現象を再現するうえではハードウェアやソフトウェアの版数(バージョン)の組み合わせなども重要になります．また，組み込みシステムの場合，個体不良もたまに起こりうるので，同一の事象がほかの装置でも起こるかどうかということに注意を払わなければなりません.

2 全体で100％になるイメージを描く

現象を再現できたら，問題の切り分けを進めていくことになるのですが，そこでのポイントは大きく二つに分かれます.

まず，全体で100％になる絵を自分の頭に描くことが必要となります．これはどういうことかというと，エラーにかかわる機能ブロックをすべて自分の頭の中に描き，それらの関係を明らかにしたうえで，どのように絞り込みをかけていけば切り分けが進むのかというイメージを描くことです.

● データの「流れ」でシステム全体をとらえる

イメージを描く場合，「流れ」で全体をとらえることがコツだと思います．例えば音響装置であれば，「入力信号が処理されて出力される」という一つの流れがあるはずです.

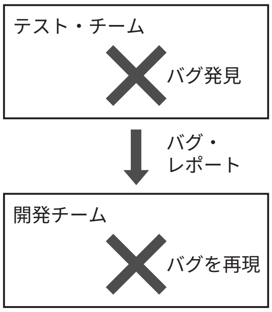


図2
バグ・レポートを受け取ったら
バグ・レポートを受け取ったら，まずそのバグを再現させること．現象の再現も試みずにゴネるのは問題解決を遅らせるだけ．深刻なバグほど「そんなはずはない」という気持ちが先に立ってしまうものだが，まず再現させて切り分けを進めないと，ほんとうは自分に非がなくても周りが納得してくれない.

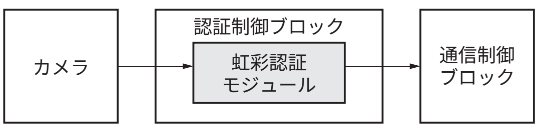


図3　虹彩認証システムの例
カメラから画像を取り込み，「認証制御ブロック」と呼ばれる機能ブロックで認証し，「通信制御ブロック」から認証情報を通信するものとする．虹彩認証アルゴリズムは真ん中のブロックにあるのだが，単体モジュールの問題の切り分けでは「全体像」と「流れ」をつかむことが重要となる.

映像を扱う装置でも扱うデータが画像になるだけで，この基本的な流れは変わりません．

図3に，Aさんが開発したシステムの概略図を示します．認証制御ブロックの中に認証モジュール，すなわちこのシステムのかなめである虹彩認証アルゴリズムが実装されているとします．左のカメラ・デバイスから映像を取り込み，認証制御ブロックで解析を行った後，そのデータは通信制御ブロックで外部機器との通信に用いられると考えられます．

このようにデータの流れでシステム全体をとらえることで，考えの偏りによる漏れをなくすことができます．ソフトウェア技術者にはありがちなことですが，例えばハードウェアに原因があってエラーが生じているのに気づかず，切り分けにとまどうことなどもあります．ソフトウェアのことが頭の中のほとんどを占めているため，どうしてもハードウェアについては考えから抜け落ちてしまいがちです．この世にあるソフトウェアは，そもそも組み込みソフトウェアでなくても，実際にはすべてハードウェアのうえで動いていることを忘れないようにしましょう．

● 過去の事例にとらわれない

また，デバッグする際，過去の成功事例（つまりスムーズに切り分けを行えた），あるいは逆にとても苦しんだ事例など，本人にとって印象深い経験があると，そこにしがみついてしまう人がいます（**図4**）．そのような状況に陥ることなく，原因がわかりにくいものであっても切り分けを進めていけるようにするためにも，上述の「全体で100％になる絵を描く」とか，「流れでとらえる」という方法を生かしましょう．とにかく，過去の事例にとらわれないということが重要です．

今回の例でいえば，例えば過去にシステムがハングアップしたとき，その原因があるモジュールのメモリ・リークであった経験があるとしましょう．その場合，今回もなにはなくともメモリ・リークを疑って，メモリ・リークがないかと部下を動員してソース・レビューを始めてしまう人がいたら，このパターンに陥っているといえます．結果的にメモリ・リークが原因であれば，それが問題解決の最短経路になるのですが，そうでなかった場合は，行き詰まることになってしまいます．

3 YESかNOかで答えが出る設問を積み重ねる

全体像が明らかになったら，次に事象の絞り込みに移ります．ここでのポイントは，「“YES”か“NO”かで答えが出る設問を積む」ということです（**図5**）．言い換えると，客観的な証拠を積み重ねて事象を絞り込むようにすること，だれが見ても原因箇所が狭まっていくとわかるように切り分けを進めるということです．

前述のメモリ・リークの例で考えてみましょう．例えば，ソース・レビューを行った結果，「ソース中のmemset，memcpyはすべて転送サイズが正しかった」ことが確認できたとします．これで「メモリ・リークはない」と断言できるでしょうか？ メモリ・リークはmallocのサイズをまちがえたときにも起こるし，引き数にとる構造体の型が関数間で不整合を起こしていた場合（例えば，ソースの版数管理を誤ったとき）などでも起こりえます．「memset，memcpyは正しい」とだけわかっても，「ハングアップは起こらない」というところまで事象を絞り込むことはできま

●XXX1年9月10日
スタック・オーバフローが原因でハング
（入社5年目の武勇伝）
●XXX4年10月1日
配列のオーバランでハング
（入社8年目の武勇伝）
●XXX6年4月10日
またもメモリ・リーク？
（現在入社10年目）

図4　デバッグは客観性がたいせつ

組み込みシステム開発に何年も携わっていると，だれでも武勇伝（徹夜など，さんざんな目に会いながらデバッグした経験）を持っている．その記憶があまりにも強烈だと，どうしてもそれにとらわれがちになる．過去の武勇伝に他人を巻き込まないこと．

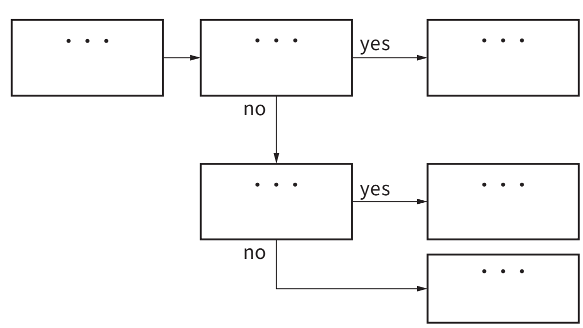


図5　切り分けはYES/NOを客観的に判別できるように絞り込む

まず，“YES”か“NO”かで答えが出る設問を立て，問題のシステムを実際に動かしてその答えがどちらに導かれるかを確かめていく．それらの問いが図のようなロジカル・ツリーに従って頭の中にスムーズに描けると，事象がおのずと絞り込める．

せん.

● 客観的な設問を考えてみる

しかし例えば,「認証モジュールをダミー・ルーチンに変えたらどうなるのか?」と考えるようにしたらどうでしょう(**図6**).この設問は答えが“YES”か“NO”かで客観的に導かれるものです.ダミー・ルーチンに変えてもハングアップが直らないという結果が導かれれば,少なくとも認証モジュールは犯人リストから外せそうです.

認証ブロックがなくなると待ち時間の整合がとれなくてシステムが正しく動かないのなら,十分に処理時間をかせげる空ループを埋め込むなどという補完手段があるでしょう.空ループが問題発生時以外のほかのデータでシステムを止めないのであれば,問題発生時の入力データでも同様に問題がないといえるからです.

この設問は,逆に「ハングアップが起こらなくなった」という場合でも同じように切り分けを進められます.なぜなら,認証モジュールがないとハングアップがなくなってしまうのであれば,このブロックが犯人である疑いが濃厚となるからです.

客観的な設問をきちんと立てることができれば,最初からメモリ・リークをにらんで怪しいソース記述を見つけられず行き詰まってしまうような場合より,ずっとスムーズに問題の切り分けを進められます.

4 自分のブロックが原因かどうかを判定

“YES”か“NO”かで答えが出る設問を作る場合も,全体から部分へと進められるように設問を組み立てていくことがポイントとなります.認証モジュールをダミー・ルーチンに変える例でいうと,認証モジュールが犯人であることが濃厚となったとき,次に試すのは,認証モジュール単体でハングアップが起こせるかどうかです.

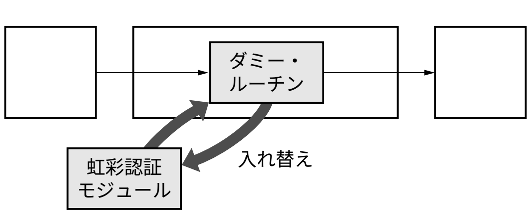


図6 良い設問の例

認証モジュールをダミー・ルーチンに変え,ハングアップの犯人がこのルーチンかどうかを確認する.「ダミー・ルーチンに変えたらどうなるか?」というのは客観的な設問の一例となる.もちろん,このようなデバッグ・コードそのものにミスがあったら手戻りが発生するので,注意しながら進めること.

●「デバッグを考えた設計」=「切り分けしやすい設計」

まず試みるのは,実際の製品から入力データを入手することでしょう.そして,入力データを認識モジュール単体に与えた場合,ハングアップが起こるかどうかを見ます.ハングアップが起こればアルゴリズムになんらかの問題があったことになるので,次はどこの関数がハングアップしているのかを特定していけばよいことになります.

実システムからこのようにモジュールの入力データを取り出すのはなかなかたいへんなことで,そのためのデバッグ・コードを作り込んでおかなければならないことも少なくありません.しかし,もしシステムが入力データを絶対に取り出せないしくみになっていたとしたら,これはもはや設計そのものに問題がなかったかを検討する必要があります.とくにこの事例のような認証システムにおいては,認証に用いるモジュールのアルゴリズムとその実装が最重

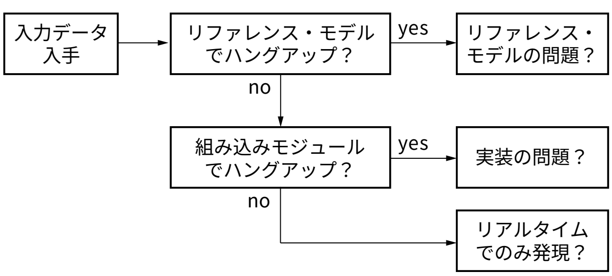


図7 単体モジュール以降の切り分け

大まかにいうと,「リファレンス・モデルに問題がある」,「組み込みモジュールに問題がある」という二つを最初に切り分けたい.後者なら次にロジックに問題があるか,リアルタイム特有の問題かを切り分ける.

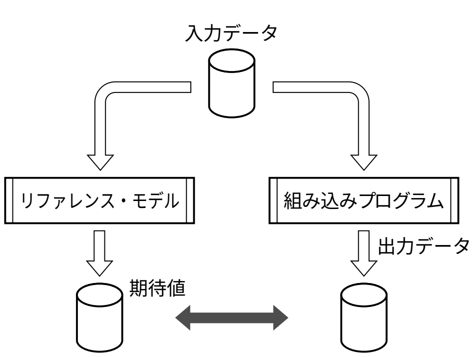


図8 リファレンス・モデル

なんらかのアルゴリズムをリファレンスとする場合,リファレンス・モデルから組み込みモジュールを実装することになる.そのため,両者がつねに同じ動きをするのが基本となっている.本文のハングアップの事例でも,両者に差分があるかをまず見極めることになる.

要部品です．したがって，この部分の切り分けができないような設計にしてしまっていたとしたら，そのことそのものが問題にされてしかるべきです．

● リファレンス・モデルとの比較が基本

また，生体認証モジュールのように，なんらかの科学法則に基づいた方式にのっとって作られたモジュールの場合，かならず「リファレンス・モデル（方式シミュレータと呼ぶこともある）」というパソコンなどで動くC言語のプログラムが用意されているものです．このようにリファレンス・モデルを実ハードウェア上にポーティングして単体モジュールを作り込んだうえで，システムに結合するというのが通常の手順です．単体の組み込みモジュールを作る場合，同じ入力に対してかならずリファレンス・モデルの同じ出力を得ることが基本となります（**図7**，**図8**）．

そのため，入力データを入手したら，このリファレンスの処理結果と比較することになります．もしリファレンス・モデルもハングアップしてしまうようなら，そもそもリファレンス・モデルのアルゴリズムに問題があることになります．そのため，どんなにがんばって組み込みモジュールを作ってもハングアップの原因は潜在していたことになります．かりにリファレンス・モデルがハングアップしないにもかかわらず，組み込みモジュールがハングアップしたのであれば，組み込みモジュールの作り込みに問題があったことになります．

また，切り分けを進めた結果，実は入力データのフォーマットがなんらかの原因で崩れていて，認証モジュールがハングアップしていたということがわかることもあるでしょう（**図9**）．そうすると，認証モジュールの前段の処理に実は問題があったということが判明します．

● リアルタイム性にかかわる部分は切り分けがやっかい

かりに，入力データでは単体モジュールがハングアップしなかったとします．その場合，この先の切り分けがめんどうです．なぜなら通常，単体モジュールを確認するときは組み込みOS上でリアルタイムに動かせません．そのため，動作確認の際に単体モジュールではハングアップが起きなくても，リアルタイム処理特有の問題（例えば割り込み耐力の有無，処理速度が限界を超えていないかなど）を洗い出せないからです（**図10**）．ひょっとすると，割り込み動作を実行しているときにのみ発生するハードウェアのバグにひっかかっているのかもしれません．あるいはもっと単純なケースで，OSに実装したときのメモリ配置と単体モジュールをテストするメモリ配置が異なっていて，単体では現象が隠れているだけかもしれません．

このようなときはどうするかというと，考えられる問題をできる限り多く挙げ，順序良く確認を進めるしかありません．メモリ配置が違うのであれば，システムとそろえて

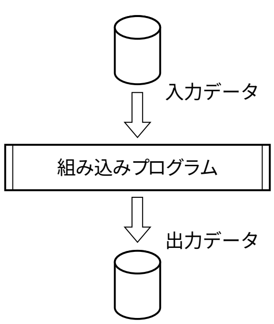


図9　ファイル入出力を用いた動作確認

単体モジュールがハングアップの原因とわかれば，単体での動作確認に移る．まずは，ファイル入出力を用いた動作確認を行う．これは，リアルタイム性を問われないテスト環境でプログラムのロジックに問題がないか（ハングアップするようなコードが含まれていないか）を確認するために行う．

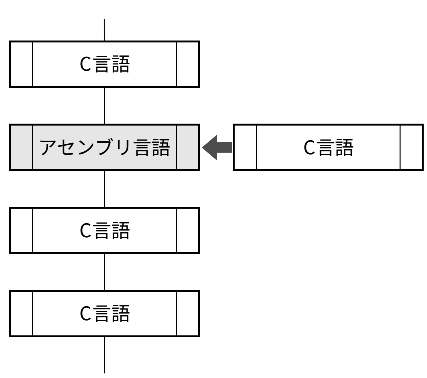


図11　一部の関数に対してアセンブリ言語による最適化がなされていた場合

最初は高級言語（多くの場合はC言語）で実装してアセンブリ言語に変えるのが常とう手段．したがって，処理速度が間に合うのなら，元の高級言語による実装に戻してみて確認する．

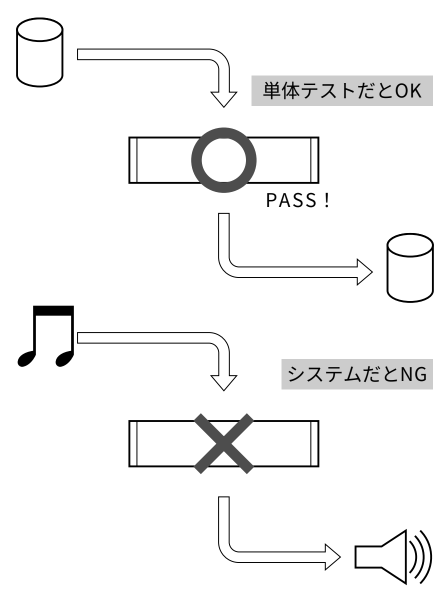


図10　リアルタイムでの単体テスト

ファイル入出力を用いたテストで問題なく動作したら，次にリアルタイムで動作する単体テストを試みる．リアルタイムでのみ発現するようなエラーは，実は切り分けが困難．「割り込み耐力がない」，「処理速度が間に合わない」，「ハードウェアのバグにつかまった」など，さまざまな要因が考えられる．一つ一つ切り分けるしかない．

単体テストをやり直すことも必要になります．問題の入力データだと実は不正アドレスにアクセスしていて，それが単体だと気づかないなどという場合には，このケースが多いと思われるので注意が必要です．処理速度がオーバしたかどうかは単体テストでも確認できるでしょう．

リアルタイム処理特有のハードウェアのバグにひっかかっていると疑われる場合，コンパイル・オプションを変えるとか，コンパイラの版数を落とす（または上げる）といったことで現象が変わることもあるので，試してみるとよいかもしれません．アセンブリ・コードをC言語に置き換えたらハングアップしなくなる，ということもありえるからです（**図11**）．

● ロジカルに「自分が原因ではない」ことを導き出す

最後に，システムにおける切り分けで重要な目安となるのは，「自分のブロックが原因かどうか」を判定するということです．要するに，自分の担当ブロックのせいなのかそうでないのかを客観的な証拠に基づいて切り分けていくということです．

これは，前述した「私が原因であるはずがない」と言い張って全体の足を引っ張るやりかたとは異なることに注意してください．客観的な証拠を積み重ねて最終的に「自分の担当ブロックが原因とは思えない」ということが見えてきたとすれば，結論を導き出すために踏んだ切り分けのステップやそこから得られた証拠は，ほかの機能ブロックを作っている人にとっても役に立つ情報になっているはずだからです．

図12に，今までの切り分けの経緯をまとめました．客観的な証拠を積み重ねながら，「自分のブロックが原因かどうか」を基準に切り分けを進めるイメージをつかんでいただきたいと思います．

5 あらかじめ切り分けの手段を整理する

システム開発に慣れた人は，バグの症状ごとに最初に試してみる手段をあらかじめいくつか用意しているはずです．認証アルゴリズムの誤認識が疑われる事象についてはリファレンス・モデルとの比較を，ハングアップに関してはダミー・ルーチンでのコールを，といった感じです．

ある程度パターン化できる切り分けの方法を論ずるために，もう一つ別のシステムを取り上げてみましょう．認証システムとはまったく別のシステムに対して，上述した考えかたを当てはめてみます．

ここでは，音響装置の例を挙げてみます．**図13**に示すように，「ディストーション」，「リバーブ」，「ノイズ・ゲート」といった複数のエフェクト・モジュールが直列に接

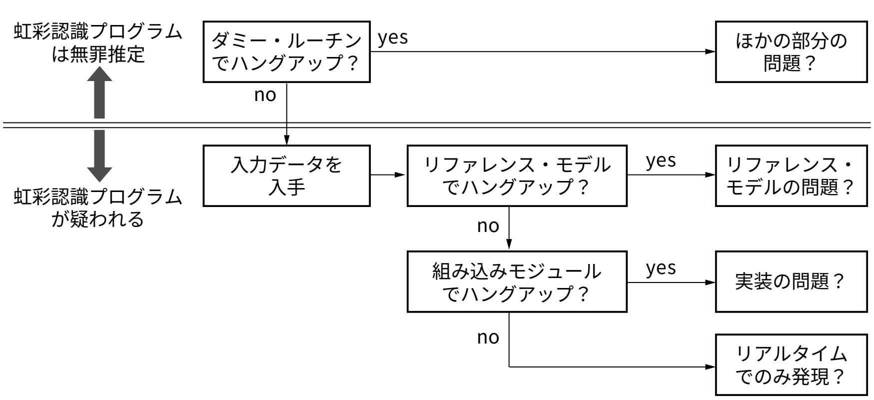


図12　ハングアップ事例の切り分けの全体像
まず，「自分のブロックが原因かどうか」を早期に切り分ける．自分のブロックが怪しいようなら，方式と実装のどちらの問題かの切り分けに着手する．後者なら，リアルタイム性の問題かどうかを確認する．客観的な証拠をロジカルに積み上げて以上の事がらを検証する．

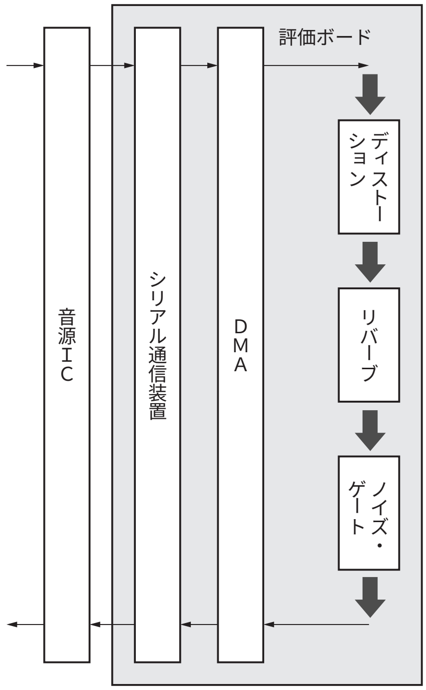


図13　あるエフェクト装置の例
ミュージシャンの演奏時にステージに並んでいるような，さまざまなエフェクト（音楽効果）を付けられる装置を思い起こしてほしい．実際にはもっと複雑と思われるが，ここでは説明のため，やや構成をシンプルに示している．

続されており，それぞれがソフトウェアで実装されているとします．入力と出力はなんらかのシリアル通信装置とDMA（direct memory access）が同期して通信するしくみをとっているとします．それらは評価（メイン）ボード上に実装されており，音源ICはそこにサブボードとして装着されているとします．

ちなみに，**図13**の「ディストーション」というのは，エレキ・ギターなどでよく聞くひずんだ音を作るエフェクトのことです．「リバーブ」はお風呂に入ったときのようなエコーを思い浮かべてもらえればよいと思います．「ノイズ・ゲート」というのは，演奏されていないときに発する電子楽器のノイズ（ジーという音）を消すエフェクトです．

あなたの担当は図の三つあるエフェクト・モジュールの実装とテストだったとします．そのような場合，システム・テスト中にバグ・レポートが上がってきたらどう対応すればよいのでしょうか．

● ハングアップしたらダミー・ルーチンを使う

ハングアップした事象については，虹彩認識プログラムの場合と同じように，空ループのようなダミー・ルーチンに取り替えて試してみる方法がここでも使えそうです（**図14**）．エフェクト・モジュールが三つあるので，順番に取り替えていけばそれでOKです．

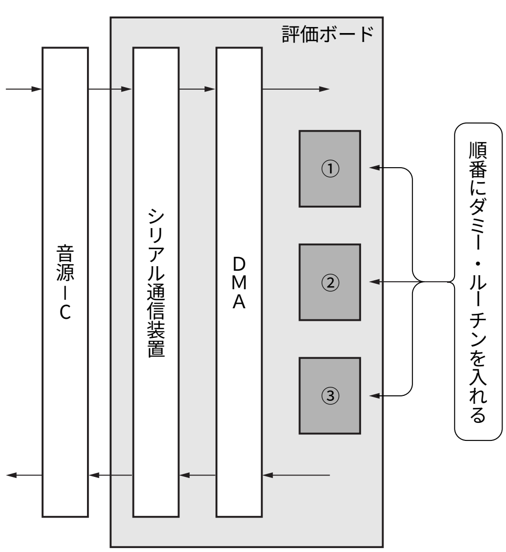


図14　ハングアップの場合の切り分け方法

虹彩認識プログラムの場合と同じく，適当な順にダミー化する方法が使えそう．ここでは，エフェクト・モジュールがソフトウェア実装されているので，これらを単純にダミー化すればよい．もしハードウェア実装ならモジュール部分のみダミー化できるテスト・ロジックが必要となるかもしれない．

● 音が出ないときは機能ブロックの境目に注目

次に，「音が聞こえなくなった」という症状を考えてみましょう．ある特定の操作を続けた結果，音が聞こえなくなるというバグ事例が上がってきた場合を考えることにします（**図15**）．

この場合もまず，再現を試みます．全体像はすでに明らかなので切り分けに移ります．ダミー・ルーチンを用意すればよいのですが，**図13**の構成であればもっと確実に“YES”か“NO”かが明らかになるポイントがあります．それは，サブボードとメイン・ボードの境目にあるシリアル通信回路と音源ICの間の信号線です．ここをロジック・アナライザなどでプローブして信号の流れをとらえることで，より確実に原因ブロックを特定できるはずです（**図16**）．

「音が聞こえない」ということは，信号がそもそも入ってこないのか，出ていかないのかの二つに一つです．エフェクト装置と音源ICの間の信号線をプローブすれば，どちらの現象が起こっているのかが明らかになります．音源ICからエフェクト装置に信号が来なければ前者，来ているのにエフェクト装置から音源ICに信号が返らなければ後者となります．前者であれば，音源ICにすでに問題があるケースです．逆に，楽音信号が入力されているのにもかかわらず，正常に出ていかない場合は，評価ボード側でなんらかの問題が生じていることになります．

●「協力しつつ裏切る」作業がデバッグ

最後に，まとめとしていくつか指摘しておきたいことがあります．現在，商用利用されているような巨大なシステムのデバッグは，複数の開発チームによる共同作業です．この過程はいわば，1枚のパズルのピースを埋める作業にたとえることができます（**図17**）．デバッグ作業に携わる人たちひとりひとりが共通の大きな1枚の絵（システムの全体）を頭に描きながら，自分が担当する部分のピースを埋めて

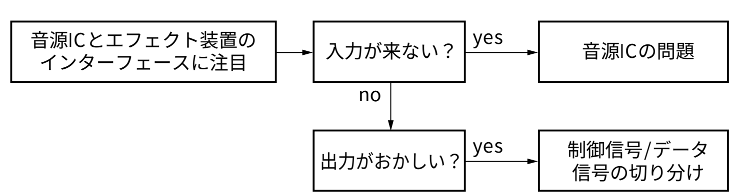


図15　音が聞こえないときの切り分け

「音が聞こえない」ということは，要するに信号が「入ってこない」か「出ていかない」かなので，どちらの事情かをまず切り分ける．そのためのデバッグのポイントは図16に例示している．

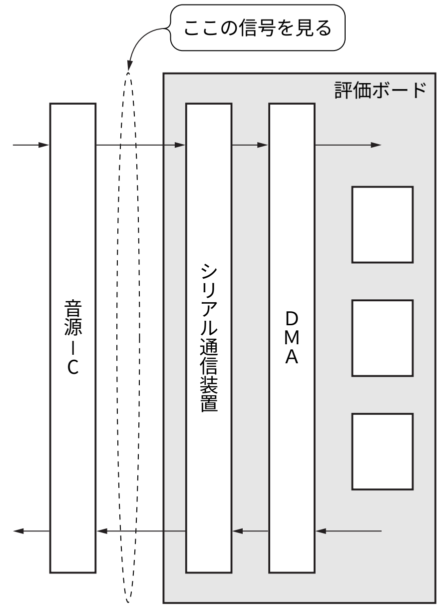


図16　音が聞こえないときのデバッグのポイント
「どのブロックに問題があるか」を突き止めることがデバッグでは重要．したがって，ハードウェア，ソフトウェアを問わず，機能ブロックの境目が重要なデバッグのポイントとなる．つまり，「入ってこない」か「出ていかない」かのデバッグのポイントは，ボードの境目になる．

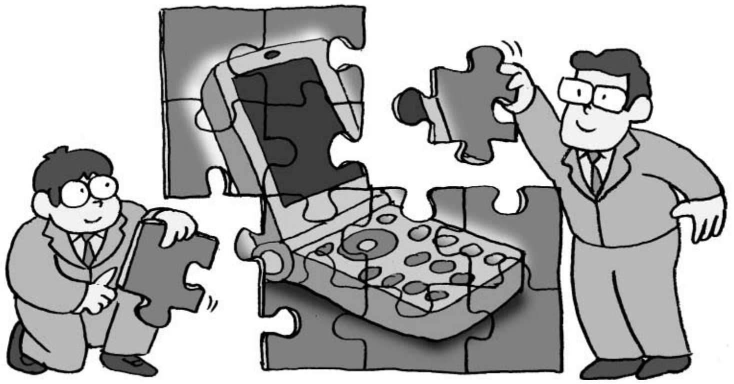


図17　1枚のジグソー・パズルを作っていく感覚
チーム開発におけるデバッグ作業は，1枚のパズルを埋める作業に似ている．これは，周りの手が見えていないとしごとが進まない共同作業であると同時に，自分以外のだれかを効果的に追い詰めなければならない作業でもある．「協力しつつ裏切る」のがゲームのルール．

いくのです．そのため，全体像から自分の担当する部分をとらえ切れていない人がひとりでもいると，ほかの人たちもピースをうまくはめ込むことができないのです．

　また，システム開発におけるデバッグ作業というのは，「協力しつつ裏切る」作業という側面も持っています．あるブロックで切り分けが進んで「自分たちが原因ではない」とわかったら，その情報はまだ切り分けの済んでいないほかのブロックの人にとっては，外堀が1ヵ所埋まってしまったことにつながるからです．もっともこの情報は，正しく提供されていればバグを抱え込んでいたブロックにとっては有益な情報になることは前述したとおりです．判明したとおりに情報をなぞってより深く掘り下げると，かなりスムーズに答えにたどり着けることもあります．優れた技術者ならそういう情報の出しかたをするように心がけるものだと思います．

さえき・はじめ

<筆者プロフィール>
冴木 元（ペンネーム）．システム・エンジニア．組み込みソフトウェア開発に携って何年もたつが，ハードウェア実装の実務経験はほぼ皆無．にもかかわらず，原稿を読んだ本誌の編集長は本人に直接会うまで，筆者がハードウェアの技術者だと思い込んでいたらしい．このでき事で少し自信がついたとのことでした（なぜか伝聞形）．

Column “文系”，“理系”にかかわらず優れた技術者になるためのアドバイス

このコラムは，組み込みシステム業界に入ってくる「文系出身」の新人エンジニアの方へのメッセージです．「文系出身」で「組み込みシステムの開発」に携わるという経歴を持つ人は，世の中には少なからずいます．筆者も実はその一人です．こうした一見矛盾（？）するような道を歩んでしまった人たちに先輩からのアドバイスを，というのがこのコラムの目的です．

● 文系，理系は関係ないが“適性”はしっかり見定めて

文系と称していながら，「ほんとう？」と疑いたくなる専攻科目の例を以下に挙げます．

- 数理経済学
- ファイナンス工学
- 社会科学全般で使われるゲーム理論

もし読者の方が文系で社会科学系を専攻していれば，こういう専攻科目があったことくらいは記憶にあるかもしれません．例えば，ゲーム理論は自然科学からの借りものではない，社会科学特有の数学理論として知られているものですが，使いこなすには当然優れた数学的センスを要求されます．上に挙げた例は社会科学系のものですが，人文系にも理系っぽいものはあります．例えば，米国でとくに盛んな「分析哲学」という学問は，哲学と称していますが高度な論理学の知識が不可欠だったりします．

このように，ひとくくりに「理系」とか「文系」とかいっても細かくみるといろいろあります．

次に，自分の適性をしっかり見定めておかないとやはりうまくいかないという例を以下に挙げてみます．

- 経済オンチの数理経済学者
- 相場感のないファイナンス・エンジニア
- そのへんのオヤジでも言えそうなことをゲーム理論で解説する大学教授

例えば，いくら数学的センスが優れていても，まるで使えない金融モデルしか作れないのなら，ファイナンスのエンジニアとしてはあまり役に立たないでしょう．ゲーム理論にしたところで，複雑な定理を証明したあげくに導かれる結論があまりにもありきたりのものであれば，そんな本は読む気をなくしてしまいますよね．

もちろん，適性を誤ればたいした成果が得られないのは，学問の分野に限ったことではありません．今年，新入社員となったみなさんは，一昨年から昨年にかけて就職活動されたでしょうから，面接を通して自分の適性を見る機会があったと思います．人と接することでみずからの適性を見出すことは，社会人になってからは非常にたいせつです．文系でありながら組み込みシステムの開発に携わることになったという読者の方は，そういった経験をされたことと思います．

昨年の今ごろ，「外資系証券会社の面接は性に合わなかった．たとえヒルズ族になれなくても，自分にはメーカが向いている」と考えていた読者の方がいたら，案外と将来は「勝ち組」になっているかもしれませんよ．

なお，上に挙げた例は細部を眺めると大枠の分類が怪しくなってくる例です．本コラムの目的は，各論を持ち出して総論の二分法を否定することではありません．むしろ，マクロとミクロの両方の視点を自在に移動させたときに何が見えてくるかに注目してみたいと思います．

●「視点の移動」と組み込みソフトウェア開発

さて，「文系出身の組み込みエンジニア」の例がそうですが，ミクロの視点で見たことがマクロの分類と矛盾するように見えることはよくあります．このようなものを目の当たりにしたとき，どのように考えてみるのがよいのでしょうか？

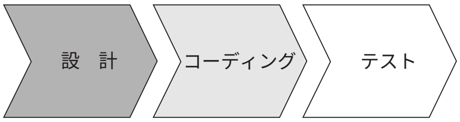
例えば，上述の「ファイナンス工学って文系？」みたいに，こうした各論を持ち出して総論の分類を否定しようとする方法は，単純な二分法の矛盾をつくためによく使われることがあります．このようなやりかたを思いつくのは簡単です．しかし，二分法をつき崩すための議論の組み立てかたとしては，あまり功を奏するとはいえません．というのは，世の中には，ミクロの視点で見たことがマクロの視点で見たことと矛盾するような例がいくらでもあるからです．

ミクロのとらえかたとマクロのとらえかたが一見矛盾する場合，たいせつなのは両者の視点を持てるようにすること，ミクロとマクロの二つの視点を自由に行き来できる目を持つことだと筆者は考えます．この「視点の移動」という考えかたを，組み込みシステムの実際の開発に当てはめて考えてみましょう．

新人エンジニアの場合，しごとに就いたらとりあえず「目の前のモノを動かすこと」が求められます．要するに，プログラムを作って正しく動作させたり，適切な動作確認を行ったりしなければならないということなのですが，組み込みシステムの場合，これが最初はなかなか困難です．話をソフトウェアに限定しても，組み込みシステムの世界は，パソコン上で開発が完結する通常のソフトウェア開発とは勝手が違うところが多々あります．

例えば，C言語の基本として知られる『“Hello World”と`printf`を実行するだけのプログラム」を考えてみましょう．こんな簡単なプログラムでも，予備知識がなければ動かすのに1ヵ月くらいかかるかもしれません．なぜかというと，組み込みシステムの開発環境においては，プログラムを動かす前提となる環境の確立に時間がかかることが多いからです．例えば，メモリ配置の問題など，パソコンでの開発ならOSやコンパイラが勝手にやってくれるはずの環境の設定を開発者が指定しなければなりません．

環境設定以外にも，CPUのアーキテクチャなど，無視できない要素がいろいろあります．対象となる組み込みシステム開発に必要となる環境やハードウェアのアーキテクチャをしっかり押さえないと，小さなプログラムであっても高性能，高品質なものは作れません．



図A-1　技術者にはマネージメント力が必要

「製品をリリースするまでの工程をいかに組み立てるか」というのは，マネージメントの基本となる．ここでは，それぞれの成果物を実際に作った経験が不可欠となる．この経験はプロジェクト・マネージメントの教科書をいくら読んでも身につかない．



図A-2
プレイング・マネージャを目ざす
サッカーのテレビ中継は，上空からの鳥観図を視聴者に見せる．優れたサッカー選手はピッチを走り回っていても，この「目」で試合を見渡せるという．みなさんも「ゲームを組み立てる」ことのできるエンジニアを目ざしてください．

●「プレイング・マネージャ」を目ざしてみよう

開発環境やアーキテクチャの知識は，コーディングなどの実作業を行うエンジニアだけでなく，実はエンジニアをまとめるマネージャになった場合にも無視できない事がらです．

入社した当初はあまり耳にしませんが，数年の経験を積むとたいていの人が「○○さんは技術力はあるが，マネージメント力がない」という評価を聞かされると思います．ここでいう「マネージメント力」というのは，開発の工程を考え，タイム・スケジュールを組み，人や物，金を割り振るという管理の技能のことです(**図A-1**)．筆者の経験に照らせば，「技術力はあるが，マネージメント力がない」という人は確かにいます．しかしその逆の例，すなわち「マネージメント力はあるが，技術力がない」という人には，この業界ではお目にかかったことがありません．これは，そもそも技術力がない人はマネージメントを行う機会すら与えられないということもあるでしょう．しかし，根本的な原因は，やはり技術が十分に備わっていない人は開発のポイントを押さえることができないからだと筆者は考えます．

「技術力」と「マネージメント力」というのは一見矛盾するようですが，両者を結びつけるように努力しないと，とくに後者は身につきません．「目の前のモノを動かす」ことのじょうずなエンジニアになることを目ざす一方で，実際に作業を進めるうえでのポイントがどのようなことかを見抜き，それを人に伝えたり計画に反映したりする技能を同時に磨いていかなければなりません．目の肥えたリーダ(プロジェクト・マネージャ)になるためには，新人のころから培った「目の前のモノを動かす」経験が不可欠なのです．

おわかりでしょうか？「技術力」と「マネージメント力」は互いに別々の事がらでもなければ，ましてや「管理する人」と「管理される人」に最初から分類してよいものでもありません．両者を結びつける努力を怠らなかった人だけが，優れた技術者として管理能力を発揮することができるのです．

このことを，たとえ話で説明したいと思います(**図A-2**)．サッカーのテレビ中継を見る機会は多いと思います．「サッカーに興味がない」という読者でも，サッカーのテレビ中継をまったく見たことがないという方はまれだと思います．

このような番組は，たいてい空撮による鳥瞰図を中心に映像を組み立てています．これはあるサッカー番組を見ていて筆者が聞いた話なのですが，優れたサッカー選手というのは，あたかもヘリコプターか何かで試合会場を鳥瞰しているような「目」を持っているそうです．つまり，試合をしながらあたかも視聴者がお茶の間でテレビを見ているかのように試合の全体像を見渡すことができるのだそうです．このような技量を持った選手でなければ，「ゲームを組み立てる」というような役割に就けないのだと思います．プレーヤとしての役割(ミクロ)と試合運びの組み立て(マクロ)を同時にこなせるからこそ，そのような選手はチームのリーダという役割を与えられるのでしょう．ほかのスポーツ，例えば野球などではごくまれに，選手としての技量と管理能力を兼ね備えた人が「プレイング・マネージャ」として監督兼選手の役割に就くことがありますね．

新人のみなさん，どうせやるなら「ゲームを組み立てられるエンジニア」を目ざしてみませんか？

繰り返しになりますが，まずは「目の前のモノを動かすこと」ができる技術者になってください．そのためには，新しい開発環境や開発手法，ドキュメントの整理のしかたなどを身につけていくことが不可欠です．それと同時に，作業するチーム全体の動きが読める人になっていってほしいと思います．もの覚えの良い人であっても，作業の優先順位が不明確なために重要な作業が滞ったり，頼んだことと違うことをやってしまってモメゴトの種を作るようなことが新人エンジニアにはありがちです．これはひと言でいえば，「チーム全体が見えていないから」ということが原因です．複数の人間がいっしょにしごとをする以上，日々変わっていく開発作業全体の流れを見抜く力を養わないと，いつまでたってもチーム開発をスムーズに行えるようにはなりません．

文系出身の組み込みエンジニアについて，ミクロとマクロの視点の移動という切り口でいろいろ書きました．「おもしろかった」と思っていただければ幸いです．逆に，「意味がわからない」という読者の方もいるでしょう．そういう方も，いろいろな考えかたをする人間がこの業界にはいるんだ，ということだけ覚えておいてください．

みなさんのご活躍を期待してやみません．